

ARRAY LOOKUP TECHNIQUES: FROM KEY-INDEXING TO HASHING (PART 2)

Paul M. Dorfman

Citibank Universal Card Services
Jacksonville, FL

ABSTRACT

Arrays have a variety of uses in the SAS ® Language. In this paper, they are viewed as a powerful DATA step structure whose direct addressability is ideally suited for quick data lookup. In this tutorial, we shall consider two array search techniques based on direct array addressing.

1. Key-indexing -- fastest existing lookup scheme for integer keys falling into a limited range. More importantly, key-indexing forms the core idea on which hashing is based.
2. Hashing -- generalization of key-indexing that allows performing direct-address lookup on keys of any length and type. In the hashing section, the following issues will be discussed:
 - Hashing in general and possible ways of implementing it in SAS.
 - Choosing an appropriate hash function.
 - Resolving collisions between keys mapping to the same address.
 - Practical DATA step implementations of hashing: Coalesced chaining and open addressing with double hashing.
 - Hashing keys that cannot be converted to a SAS numeric variable without losing precision and non-natural keys.

Lookup techniques presented in the tutorial are demonstrated using a real-life example of matching two files by a common key. The methods are benchmarked against each other and other lookup methods, such as "big formats" formats, in terms of resource efficiency and performance.

INTRODUCTION

Table lookup is one of the most frequent operations performed in data processing, and SAS provides us with a rich collection of built-in searching techniques. Merging, SQL joins, formats, SAS indexes, to name a few, all serve the purpose of looking up relevant data. In addition, SAS Language incorporates arrays - data structures ideal for implementing just about any searching algorithm. SAS arrays are not ready-to-go tools: Array-based table lookups have to be hand-coded and tuned. However, this "sniper" approach is more flexible and often results in programs that search faster and use fewer resources than the "heavy artillery" does.

To make the discussion less abstract, we will consider a common task of matching two SAS data files, SMALL and LARGE, by a key variable. This will enable us to see how different array lookup techniques compare to each other and to some of the ready-to-go methods.

In Part I presented in the Beginning Tutorials section, we considered lookup techniques based on comparisons between keys and *finished* with key-indexed search, the purest form of table lookup based on direct addressing. Key-indexing is the fastest known search method, but it has limited applicability. In this paper, we will *begin* the discussion with key-indexing. We will see that the core idea of key-indexing can be generalized to simultaneously remove its limitations and preserve its stunning performance, and

lead to a powerful hybrid method called *hashing*. The rest of the paper will be devoted to the discussion of this fascinating technique, its implementations and usage in SAS.

THE SAMPLE PROBLEM

Consider the following common data processing task. One SAS data file called SMALL contains a variable KEY, not necessarily unique, and a satellite variable S_SAT. Another file, LARGE, contains a variable KEY of the same type and length as SMALL, and may also have some satellite information L_SAT. Both files contain, generally speaking, random data, and are unsorted. *For the time being, let us assume that the key variables are non-negative integers.* The files can be simulated, for example, like this:

```
%LET N_SMALL = 1E5;
%LET N_LARGE = 1E7;
%LET N_RANGE = 1E9;

DATA SMALL (KEEP=KEY S_SAT)
  LARGE (KEEP=KEY L_SAT);
DO I=1 TO &N_SMALL;
  S_SAT ++ 1;
  KEY= CEIL(RANUNI(1)*&N_RANGE);
  OUTPUT SMALL;
END;
DO I=1 TO &N_LARGE;
  KEY= CEIL(RANUNI(2)*&N_RANGE);
  L_SAT ++ 1;
  OUTPUT LARGE;
END;
RUN;
```

The value of N_RANGE thus determines the number of digits in KEY. Let us imagine that on one hand, LARGE is so, well, large that sorting is an extremely unwelcome option, but on the other hand, we have enough memory to hold all the keys from SMALL at once. Given these conditions, *What is the most efficient way to subset LARGE based on the values of KEY in SMALL*, in other words, to match SMALL and LARGE by KEY producing a file MATCH?

Many solutions are possible. For example:

1. Sort SMALL by KEY, remove duplicates, store all keys from SMALL in a "large format" (how large, depends on the size of SMALL) using CNTLIN=, and search the format for every record read from LARGE.
2. Join the files with PROC SQL with BUFFERSIZE set to a large integer prompting the SQL optimizer to use the SQXJHSH access method.
3. Sort SMALL by KEY removing duplicates, load the keys in an array, and use binary or interpolation search to look it up for each record in LARGE.

All these methods have advantages and shortcomings, but at least one will get the job done. Yet, here we are going to try something else. With so many working techniques to choose from, why

bother? The answer is simple: Because there is a much faster and better way to do the trick!

FROM KEY-INDEXING...

Sequential, binary and interpolation techniques discussed in the Part I of the tutorial are all based on *comparing a search key to some keys stored in the array*. The main goal was to minimize the amount of comparisons, computations, and data movement. However, all comparison-based methods are *principally limited*. Unless the keys are distributed uniformly, none can complete a llookup in fewer iterations than binary search.

The only way to overcome the limitation is to *do away with comparisons between keys* and base searching on a radically different type of data organization known as *direct array addressing, key-indexing, or distribution*.

Imagine that all the keys are one-digit natural numbers. Hence, we know in advance that no key can assume any values other than 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Suppose that SMALL consists of the following 7 records:

KEY		2		3		5		7		9		5		3
-----+-----														
S_SAT		1		2		3		4		5		6		7

Let us create an array HKEY with one node (location, address) for each possible key value:

0	1	2	3	4	5	6	7	8	9
[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]	[.]

Now, for every record from SMALL, let us look at the array location whose index equals the value of the KEY. If the node is empty, we will move S_SAT into it:

0	1	2	3	4	5	6	7	8	9
[.]	[.]	[1]	[2]	[.]	[3]	[.]	[4]	[.]	[5]

We have prepared a *key-indexed table*, comprising two types of entries: *empty and occupied*. Note that by inserting a satellite into *only if the node is empty*, we retain just the *first instance* of the satellite corresponding to a repeating key value. Without this condition, the *last instance* would be used. Either way, all duplicate keys are removed automatically as the table is loaded.

If SMALL has no satellites or we are not interested in pulling them into MATCH, the entries of the key-indexed table could be marked as occupied by moving a non-missing value, for instance, 1, into the node. Once again, duplicate keys are removed on the fly.

From that point on, to search for a key, we simply examine the location in the table whose index is equal to the key. If the location is empty, the key is not present in the table. If it is occupied, the key is found, and the contents of the node will return the corresponding satellite into S_SAT. For example, if we are looking for KEY=1, the search will be unsuccessful because the address 01 of the table is empty. The node 07 corresponding to KEY=7 is occupied, so the key is found, and the satellite is HKEY(KEY)=4.

Notice that *searching is implicitly incorporated in the loading process*. To answer "Is the node empty?" we are in effect searching the table to find out whether the current key is present in the table. If yes, the key is duplicate, and the next record is processed. From the nature of the procedure, it is unnecessary to sort SMALL, nor it is necessary to insert the keys themselves, since effectively the keys are "inserted" by making the locations corresponding to their values occupied.

Back to the sample problem: Suppose that the keys in our test files have been created as 6-digit natural numbers by setting N_RANGE to 1E6. Therefore, the entire universe of their possible values is covered by the set of natural numbers from 0 to 999999, so we should allocate the key-indexed table accordingly. Using the

procedure described above, matching the files can be implemented as follows:

```
*** KEY-INDEXING ***;

DATA MATCH;
  ARRAY HKEY (0:999999) _TEMPORARY_;
  DO UNTIL (EOF);
    SET SMALL END=EOF;
    IF HKEY(KEY) = . THEN HKEY(KEY) = S_SAT;
  END;
  EOF = 0;
  DO UNTIL (EOF);
    SET LARGE END=EOF;
    S_SAT = HKEY(KEY);
    IF S_SAT > . THEN OUTPUT;
  END;
RUN;
```

We can plainly see from the nature of the algorithm that *no other existing lookup method is simpler or can run faster than key-indexing*. It completes any search, successful or not, without comparing any keys by using a single array reference. In addition, key-indexed search has an important property: *Its speed does not depend on the number of keys* "inserted" into the table. Regardless of whether SMALL contains 10 or 1,000,000 keys, one act of key-indexed search takes precisely the same time.

Just to get a feeling what key-indexed search is capable of, here is an experimental table. It compares the performance of key-indexing in both load and search phases to that of formatting and SQL (using SQXJHSH join method) for N_LARGE = 1,000,000 (233 MHz P-II NT workstation running SAS version 6.12; the times are given in seconds, memory usage - in megabytes). Note that in all result tables presented in the paper, memory usage is shown as given by NT task manager, so the actual memory consumption by a step is lower by 7 to 8 megabytes occupied by the SAS session.

N_SMALL	METHOD	LOAD	SEARCH	ALL	MEM
=====					
100,000	KEY-INX	0.2	4.0	4.2	20.2
	SQXJHSH	N/A	14.6	14.6	18.4
	FORMAT	15.3	22.9	38.2	17.1

200,000	KEY-INX	0.5	4.3	4.8	20.2
	SQXJHSH	N/A	18.8	18.8	19.2
	FORMAT	28.5	24.6	53.1	20.5

300,000	KEY-INX	0.7	4.7	5.4	20.2
	SQXJHSH	N/A	22.7	22.7	22.9
	FORMAT	45.0	26.1	71.1	27.1

400,000	KEY-INX	0.9	5.0	5.9	20.2
	SQXJHSH	N/A	27.3	27.3	22.9
	FORMAT	58.5	27.3	85.8	34.2

500,000	KEY-INX	0.9	5.2	6.1	20.2
	SQXJHSH	N/A	27.9	27.9	28.1
	FORMAT	65.0	28.8	93.8	40.4
=====					

As we see, key-indexing completely dominates the competition. Not only it is always more than 3 times faster than its closest competitor, it would, in fact, also run 3 times faster than MERGE if both files were already sorted.

The question is, if key-indexed search is *that* good, why not forget everything else and use it at all times? *The fly on the ointment of key-indexed search is that it is only applicable when lookup keys are integers falling in a limited range, or if they can be rapidly converted to such integers*. Our test keys are six-digit natural numbers, so there are only 1,000,000 distinct values a given key can assume, and array space can be allocated for all of them at

the expense of mere 8 MB of memory. Given enough memory, one could probably get away with seven-digit keys. But if the keys represented 9-digit social security numbers, an array with 1 billion elements would be needed, which is *almost* impossible, whilst 16-digit credit card numbers would make key-indexing unfeasible. The situation is even worse in the case of 256-radix character keys, as the decimals representing them grow beyond physically possible array sizes at only 4 bytes of length. On the other hand, in many real-world situations key data do fall in a limited integer range. In such cases, key-indexing, with its blazing speed and simplicity, is beyond competition. Here are some examples:

1. SAS date values. Any practical date range can be covered by a limited integer range. For instance, a key-indexed table sized as [-138061:380217] will account for all SAS dates up to the 4th millennium using only 4 MB of memory.
2. SAS time values. An array sized as [0:86400] will suffice to key-index any SAS time value.
3. ICD9/CPT4 codes are short, mainly digital, strings. If some character is a letter, it can be converted into a number in 1-26 range (the idea belongs to Don Stanley).
4. 1-byte alphanumeric keys will map onto the range [0:255] as RANK(KEY). INPUT(KEY,PIB2.) will map all 2-byte keys map onto the range [0:65535].
5. Any fractional keys that result in limited range integers when multiplied by a suitable constant.

Most importantly, however, the idea of direct addressing on which key-indexing is based can be *generalized to embrace keys of any type and range* resulting in a class of search methods called *hashing*. Hashing retains the performance advantages of key-indexing without requiring unrealistic memories.

...TO HASHING

Both the speed and limited applicability of key-indexing stem from the following:

1. The lookup table is accessed via direct addressing.
2. The entire universe of possible key values is addressable. That is, a separate array node must be allocated for each possible value a key may assume.
3. No comparisons between keys are made.

In order to preserve the speed of key-indexing, we should preserve the idea of direct addressing and, therefore, stay with #1. However, if we need to process *keys falling in any range*, there is no other option than to abandon the restriction imposed by #2. As it turns out, the only way to accomplish it is to part with #3 as well, by allowing certain amount of comparisons between keys. To see how it is possible, imagine that SMALL contains just 10 keys:

185 971 400 260 922 970 543 532 050 067

The keys are 3-digit, so in order to use key-indexing we would have to allocate a key-indexed array sized as [0:999] and map each key to the array node corresponding to its value. Out of 1000 table nodes, only 10 will be occupied while the memory allocated to the rest of the nodes will be simply wasted!

The question is, *can we get away with a smaller table of a reasonable size, only somewhat larger than the number of keys*, and still be able to capitalize on direct addressing? The answer is "yes". Let us choose some number HSIZE somewhat greater than the number of keys N_SMALL, for instance, HSIZE=17, and allocate a temporary array sized as HKEY(0:17). Let us agree to call the array HKEY the *hash table*, HSIZE - the *hash table size*, and the ratio N_SMALL/HSIZE - the *load factor*. Thus, the load factor shows the number of lookup keys relative to the total number of nodes in

the hash table, in other words, how *sparse* the hash table is. In our example, the load factor equals 0.588, that is, the hash table is about 41 percent sparse.

It would be magnificent if we could use some fast function H(KEY) that would take a key as an argument and return an address into HKEY, unique to each key supplied. This way, H(KEY) would map each key to its own location in HKEY in one-to-one manner. If such function, called a *perfect hash function*, were available, we would have to do nothing else but plug it in the code for key-indexed search. Such functions *are* possible, they are quite difficult to discover, and once discovered, a perfect hash function can only be used for the same set of keys.

We can obtain a much more flexible method if we give up the requirement of one-to-one relationship between the keys and table addresses and thus permit H(KEY) to *map two distinct keys to the same location* in HKEY. However, if two different keys are sent to the same hash address, a phenomenon termed a *collision* occurs, and we must invoke some *collision resolution policy* in order to tell the keys apart in the process of insertion and/or search.

Thus, we arrive at the *core concept behind hashing*. If the hash function H(KEY) is constructed in such a way that only *a few keys* get mapped to a particular hash address H, we can employ the following strategy:

1. Given a search key KEY, use H(KEY) to *hash* KEY to some address H in the table.
2. If the address H is empty, the search is unsuccessful.
3. If the address is occupied, search the keys residing at H *sequentially*.

As we see, hashing is a typical *hybrid algorithm*: It combines direct addressing with *sequential search*, a method based on comparisons between keys. The *average* number of keys mapping to any hash node equals N_SMALL/HSIZE, i.e. the load factor. Therefore, we should expect that if the hash table is not full and the hash function spreads the keys uniformly, the average number of key comparisons required to either find or reject a search key should be less than 1. We should also expect that the sparser is the hash table, the faster hashing will run. (Numerous experimental results corroborate this conclusion.). So, in order to use a hash table, we have to:

1. Choose a proper hash function H(KEY).
2. Find an efficient way to resolve collisions.

Apparently, the efficiency of hashing heavily depends on choosing a decent hash function. What properties should such a function possess? The question will be easier to answer if we find out first what a *bad hash function* would do.

On one extreme, if we had a lightning fast function mapping all the keys, or their majority, to the same hash address, it would certainly have to be considered terrible. It would defy the very purpose of distributing the keys among different addresses, since then we would have to search all these keys sequentially! Hence, it is paramount that a good hash function should map the keys evenly across all hash table nodes, without burdening some addresses with huge clusters leaving the rest of the addresses empty.

On the other extreme, if we had a function that would map the keys extremely uniformly but require slow computations, it would not be of any good use, either, since in this case, the process of direct addressing itself would become a bottleneck. Hence, we can now formulate some rational *requirements a good hash function* should satisfy:

1. Its computation should be as fast as possible.
2. It should distribute the keys uniformly to minimize collisions.
3. It must return addresses in the range from 0 to HSIZE.

There is a number of mapping techniques satisfying these requirements (for instance, see [1]). In this paper, we will discuss

just the simplest, called *the division method*, which makes use of the remainder modulo:

```
H = MOD (KEY, HSIZE).
```

It certainly fits #3, since no matter what the value of KEY is, this function always returns an integer in the range from 0 to HSIZE-1. It also satisfies the requirement #1, for although it implicitly incorporates a division, its computation is reasonably fast. However, to satisfy #2, the value for HSIZE must be chosen rather carefully. Suppose that R is the radix of the character set, and X and Y - small integers. It can be shown that if HSIZE is selected as a *prime number* such that $R \cdot X$ is not equal to Y, the keys will be spread satisfactorily. Therefore, the following procedure of selecting HSIZE can be recommended:

1. Select the *target load factor*, typically in the range from 0.5 to 0.8. Keep in mind that although smaller load factors improve performance, they lead to proportionally higher memory consumption. Depending on the collision resolution policy, the load factor can be chosen as high as 1.
2. Divide N_SMALL by the target load factor and compute the remainder.
3. Choose HSIZE as the smallest prime number greater than or equal to the result.
4. The actual load factor (in most cases just slightly greater than the target) now equals $N_SMALL/HSIZE$.

Let us see how it works for our set of 10 keys. If we choose the target load factor as 0.625 and divide it into the number of keys, 10, we will obtain 16. The first prime greater or equal to 16 is 17, so it is selected as HSIZE. The actual load factor is now $10/17 = 0.588$. We should therefore allocate a hash table as

```
ARRAY HKEY (0:17) _TEMPORARY_;
```

To obtain hash addresses, we divide each key by HSIZE=17, compute the remainder H and insert KEY into the slot # H:

```
00      .      .      .
01   970      .      .
02   971      .      .
03      .      .      .
04   922      .      .
05   260 532      .
06      .      .      .
07      .      .      .
08      .      .      .
09   400      .      .
10      .      .      .
11      .      .      .
12      .      .      .
13      .      .      .
14      .      .      .
15   185      .      .
16   543 050 067
17      .      .      .
```

As we see, the keys 970, 971, 922, 400, and 185 were mapped uniquely. However, the keys 260 and 532 produced a single collision at the address 05, and the keys 543, 050, and 067 resulted in a double collision in the node 16. If this table were to be stored in memory and subsequently searched, the collisions at the locations 05 and 16 would have to be resolved. Instead of determining HSIZE by hand and/or referring to a table of primes, HSIZE can be obtained dynamically, stored in a macro variable, and used to size a hash table:

```
%LET LOAD = 0.5;
DATA _NULL_;
  P = CEIL(SIZE/&LOAD);
```

```
DO UNTIL (J = U + 1);
  P + 1;
  U = CEIL(SQRT(P));
  DO J=2 TO U;
    IF NOT MOD(P,J) THEN LEAVE;
  END;
END;
CALL SYMPUT('HSIZE',LEFT(PUT(P,BEST.)) );
STOP;
SET SMALL NOBS=SIZE;
RUN;
```

HOW TO RESOLVE COLLISIONS?

As we already know, selecting a decent hash function is just one part of the deal. No matter how good the function is, it is *practically guaranteed* that some keys will hash to the same addresses in the table, so we have to devise a method of resolving these collisions. This is another point where hashing deviates from key-indexing. We needed not store the keys in a key-indexed table, since its nodes identified the keys uniquely, and they were not compared. To resolve collisions in hashing, the keys themselves may have to be inserted into the hash table because they will have to be compared to a given search key unless it hashes to an empty address. Different *collision resolution policies* differ in the ways by means of which colliding keys are stored, linked together (as belonging to the same address), and traversed.

Separate Chaining

One way of resolving collisions naturally suggests itself once we look at the distribution of our 10 keys among the addresses of the 17-node hash table shown in the previous section. Keys attached to their respective addresses form visible "chains" (consisting of a single key in the absence of collisions). Making use of such chains to resolve collisions is called *collision resolution by separate chaining*.

There are two ways the chains of keys can be utilized in terms of the SAS DATA step. First, the keys comprising the chains could be stored *outside the hash table* by physically placing them in the occurrences of a two-dimensional array. This way, the memory for colliding keys is allocated *sequentially*. In order to follow this path, the length of the longest chain must be predetermined using a separate pass through SMALL, otherwise it will not be possible to size the second dimension of the array exactly. However, the extra read through the file SMALL (which in reality can be quite sizable) is a minor problem. A really huge drawback of this method of chaining is poor memory utilization. If we have, say, 100,000 keys in SMALL and the longest chain consists of 10 keys (there may be just one like that!), it will be necessary to allocate a two-dimensional array sized as (0:10, 0:100000) to resolve the collisions. Even with the load factor 1, it requires 10 times the memory the keys would occupy by themselves. Another disadvantage of this approach is that it is difficult to find a non-convoluted way of handling the satellites from SMALL in the case they must be pulled along with the matching keys. On the positive side, A) the two-dimensional separate chaining is quite fast, and B) it allows load factors greater than 1. So, if good memory utilization is not a paramount consideration, the method could be recommended. (Contact the author for the details of implementation.).

However, separate chaining can be exploited in a much neater fashion! Once the philosophy of allocating the storage for colliding keys is changed from sequential to *linked*, we arrive at an extremely elegant collision resolution policy, both very fast (practically as fast as the two-dimensional chaining) and reasonably memory-efficient.

The *core idea* here is to place the chains of colliding keys into the hash table itself and combine the keys hashing to the same node into a linked list with the head residing at the colliding address. Setting the last link of each chain to null designates the

end of the chain, thus helping us tell where to stop when the list is traversed serially. Since the linked lists are thus allowed to overlap in the hash table sharing the same storage locations, this approach is termed *coalesced list separate chaining*, or, shorter, *coalesced chaining*. To make it possible, all we need is a numeric array of link items, say, LINK, parallel to the hash table. Note - and it is extremely important! - that in order for coalesced list hashing to work, *at least one entry in the table must be empty*. Otherwise if the table is full, there would be no room to place the final null link needed to terminate the traversing loop. Since a table allocated as (0:HSIZE) has HSIZE+1 entries, but the modulo-based hash function can address only HSIZE nodes (from 0 to HSIZE-1), this demand will always be satisfied. In coalesced chaining, we will agree to *always leave the address 0 empty* by hashing keys as

```
MOD (KEY, HSIZE) + 1.
```

That is, if KEY modulo HSIZE is, say, 05, it will be hashed to the address 06. As stated above, we have to allocate two parallel arrays, one for the hash table itself and one to hold the links:

```
ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY LINK (0:&HSIZE) _TEMPORARY_;
```

Now we are ready to spell a detailed plan of loading the table:

STEP 1. Set an auxiliary variable R to the top address: R = HSIZE.
STEP 2. Hash: H = MOD(KEY,HSIZE) + 1.
STEP 3. If LINK(H) = ., the node is empty, no list is attached to it. Go to step 8 to insert the key.
STEP 4. Otherwise *traverse the chain* to find if the key is already in the table:
 A. If KEY = HKEY(H) the key is duplicate. Get the next key and return to step 2.
 B. Else If HKEY(H) is not 0, it is not the end of the list yet. Set H = HLINK(H) and repeat step 4.
STEP 5. Find an empty address closest to the top: Decrement R until LINK(R) = .
STEP 6. Store the key at this address: HKEY(R) = KEY.
STEP 7. Memorize where KEY actually belongs: LINK(H) = R; H = R.
STEP 8. Insert KEY into the address H and set its link to null: HKEY(H) = KEY; LINK(H) = 0. Now the node is occupied.

Let us see what kind of linked list hash table will be concocted by this process for the set of our 10 random keys and HSIZE = 17. That should make the concept more transparent. Here is how the final loaded looks like:

H	HKEY	HLINK
00	.	.
01	.	.
02	970	00
03	971	00
04	.	.
05	922	00
06	260	15
07	.	.
08	.	.
09	.	.
10	400	00
11	.	.
12	.	.
13	067	00
14	050	13
15	532	00
16	185	00
17	543	14

The colliding keys are shown in boldface. Let us take a look at the final stage of the loaded table to reveal how the collisions are

handled. The keys 543, 050, and 067 all hash to the same address 17. The first key of the chain, 543, must be stored at this address, and there it is. The link of the address 17 is not 0, hence, it is not the end of the list. Instead, LINK(17) = 14. This is the node where the next key in the chain, 050, must reside, and it is there, indeed. But once again, the list must continue because the address 14 contains a non-zero link, LINK(14) = 13. Finally, we find the key 067 in the node 13, and it is the last key colliding at the hash address 17, for LINK(04) = 0.

Likewise, the keys 260 and 532 both hash to the address 06. Looking at this node we find that it is exactly where 260 has been inserted. However, LINK(06) points to the node 15 containing the key 532, the last key in this linked list since the node is marked by a null link, LINK(15) = 0.

As opposed to colliding keys, the keys hashing to their addresses uniquely bump in a null link at once. For example, the key 922 hashes to the address 05, with LINK(05) = 0.

At this point, it should be crystal clear how this kind of table organization facilitates searching. Suppose that we need to look for KEY = 051. This value modulo 17 is 0, therefore it hashes to the address 01. At this address, the link field LINK(01) is missing. That means that none of the keys in the table has ever hashed to this address, which immediately leads to the conclusion that the key is not in the table. However, searching for KEY = 047 that hashes to the address 14, we encounter a different situation because LINK(14) = 13 is not empty. Hence, one or more keys in the table hash to this address, too, and the entire chain must be examined for the presence of 047. Since the key 050 sitting in the node 14 does not match the search key, we have to look at the next key in the chain located at the address 13 to which LINK(14) is pointing. We find that the key 067 in the node 13 does not match the search key 047, and it is the end of the list since LINK(13) = 0. Hence, 047 is not present in the table, either.

As an example of a successful search, let us try to find KEY = 050. It hashes to the address 17 containing the key 543, different from 050. But it is not the end of story: LINK(17) = 14 is not null and tells us that the next comparison should be made with HKEY(14) = 050. At this point, the search key is found, the list need not be traversed any further, and the process terminates successfully.

After this walk-through, it should not take a Certified SAS programmer to schedule *hash searching*. In fact, it is invoked by the loading process when we traverse the chains of keys inserted so far looking for duplicates.

STEP 1. Hash: H = MOD(KEY,HSIZE) + 1.
STEP 2. If LINK(H) = . then search terminates unsuccessfully.
STEP 3. Traverse the list. If KEY = HKEY(H) then search terminates successfully.
STEP 4. Else examine LINK(H). If LINK(H) = 0 then KEY is not found. Stop.
STEP 5. Next link. Set H = LINK(H) and go to step 3.

Now we have everything we need to solve the file match problem using coalesced chain hashing. An additional array parallel to the hash table and links, HSAT, is used to pull the satellites from the lookup file SMALL. If we need the keys only, the array HSAT and instructions referring to it should be omitted.

```
*** COALESCED CHAIN HASHING ***;

DATA MATCH (KEEP=KEY S_SAT L_SAT);
  ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
  ARRAY LINK (0:&HSIZE) _TEMPORARY_;
  ARRAY HSAT (0:&HSIZE) _TEMPORARY_;
  R = &HSIZE;
  DO UNTIL (EOF);
    SET SMALL END=EOF;
    H = MOD(KEY,&HSIZE) + 1;
    FOUND = 0;
```

```

        IF LINK(H) > . THEN DO;
            LINK TRAVERSE;
            IF FOUND THEN CONTINUE;
            DO WHILE (LINK(R) > .);
                R +-1;
            END;
            LINK(H) = R;
            H = R;
        END;
        HKEY(H) = KEY;
        HSAT(H) = S_SAT;
        LINK(H) = 0;
    END;
    EOF = 0;
    DO UNTIL (EOF);
        SET LARGE END=EOF;
        FOUND = 0;
        H = MOD(KEY,&HSIZE) + 1;
        IF LINK(H) > . THEN LINK TRAVERSE;
        IF FOUND THEN DO;
            S_SAT = HSAT(H);
            OUTPUT;
        END;
    END;
    END;
    STOP;
    TRAVERSE:
        IF KEY = HKEY(H) THEN FOUND = 1;
        ELSE IF LINK(H) NE 0 THEN DO;
            H = LINK(H);
            GOTO TRAVERSE;
        END;
    RETURN;
RUN;

```

This code intentionally parallels the algorithm above, so one should not be surprised to find a GOTO instruction. Those abhorring GOTOS and favoring "structured programming" may prefer to rewrite the TRAVERSE block as

```

DO UNTIL (FOUND);
    IF HKEY(H) = KEY THEN FOUND = 1;
    ELSE IF LINK(H) = 0 THEN LEAVE;
    ELSE H = LINK(H);
END;

```

Now that a hashing routine is ready, we can test it using the sample files. Note that the requirement of limited integer range can now be dropped: the code above runs equally fast, no matter whether the keys have 6, 9, or 16 digits. However, in order to compare chained hashing with the results obtained earlier, the 6-digit range has been retained. The program was run for N_LARGE=1,000,000, and two load factors: 0.5 (50 percent sparse table) and 0.8 (20 percent). Note that the table below also contains the results obtained for open addressing with double hashing that will be discussed later on. So far, they can be disregarded. As before, the times are given in seconds, and memory usage is expressed in megabytes:

N_SMALL	METHOD	LOAD	SEARCH	ALL	MEM
=====					
100,000	KEY-INX	0.2	4.0	4.2	20.2
	SQXJHSH	N/A	14.6	14.6	18.4
	FORMAT	15.3	22.9	38.2	17.1
	LOAD=.5 CHAIN HSH	0.7	6.3	7.0	9.8
	LOAD=.8 CHAIN HSH	0.9	8.6	9.5	8.3
	LOAD=.5 DOUBL HSH	0.9	8.8	9.7	8.0

200,000	KEY-INX	0.5	4.3	4.8	20.2
	SQXJHSH	N/A	18.8	18.8	19.2
	FORMAT	28.5	24.6	53.1	20.5
	LOAD=.5 CHAIN HSH	1.4	6.3	8.1	13.4
	LOAD=.8 CHAIN HSH	1.9	8.6	9.5	11.1
	LOAD=.5 DOUBL HSH	1.8	8.8	10.6	9.3

300,000	KEY-INX	0.7	4.7	5.4	20.2
	SQXJHSH	N/A	22.7	22.7	22.9
	FORMAT	45.0	26.1	71.1	27.1
	LOAD=.5 CHAIN HSH	2.5	6.3	8.8	15.4
	LOAD=.8 CHAIN HSH	2.6	8.6	11.2	13.7
	LOAD=.5 DOUBL HSH	2.5	8.8	11.3	11.8

400,000	KEY-INX	0.9	5.0	5.9	20.2
	SQXJHSH	N/A	27.3	27.3	22.9
	FORMAT	58.5	27.3	85.8	34.2
	LOAD=.5 CHAIN HSH	2.9	6.3	9.2	17.2
	LOAD=.8 CHAIN HSH	3.4	8.6	12.0	15.0
	LOAD=.5 DOUBL HSH	3.2	8.8	12.0	14.5

500,000	KEY-INX	0.9	5.2	6.1	20.2
	SQXJHSH	N/A	27.9	27.9	28.1
	FORMAT	65.0	28.8	93.8	40.4
	LOAD=.5 CHAIN HSH	3.2	6.3	9.5	22.8
	LOAD=.8 CHAIN HSH	4.1	8.6	12.7	17.7
	LOAD=.5 DOUBL HSH	4.0	8.8	12.8	16.6
=====					

The method of chained hashing has a number of curious features:

1. The lower the load factor, i.e. the sparser the hash table, the faster the search, but it consumes proportionally more memory.
2. If the hash table is kept relatively sparse, it possesses a fascinating property: Its lookup time does not depend on the number of keys in the table. It is not accidental that the same property is inherent to key-indexed search.
3. It runs somewhat slower than key-indexing but still 2 to 3 times faster than even SQLXJHSH (which, by the way, is a hash, too!) and is much easier on memory than the rest of the methods tested.
4. Just like key-indexing, hashing needs neither sorting nor removing duplicates. The latter are removed automatically in the process of loading a hash table.

Open addressing

Hashing with separate chaining is one way of resolving collisions. It performs very well and is not very sensitive to the load factor. However, it has a significant shortcoming: It requires an array to hold links, as large as the hash table itself, and that consumes additional memory. If the file SMALL is actually large (in real-world applications, lookup tables do get millions of keys), the extra memory burden can be quite significant.

A different class of collision resolution methods called collectively *open addressing* does away with links altogether, and, therefore, reduces memory consumption, especially if satellites are not dragged through the memory and do not dominate the process.

The main idea behind open addressing can be described as follows. Just like in the list hashing, keys are stored in the hash table itself. Suppose that as we load the table it is discovered that a key hashes to a non-empty node producing a collision. Instead of inserting the key into the next available empty address and tracing its original hash address via a link, we can simply step down the table one node at a time until an empty location is found and place the colliding key there. If the bottom of the table is reached, we can wrap to its top and continue the process. Then at the time of search, if a key hashes to an occupied node, we can use the same exact path of stepping through the table until either a match is found or a node is empty. The latter would signify that the key is not in the table.

However, such a technique of resolving collisions termed *open addressing with linear probing* (called this way because the table is traversed in a linear fashion) is a very poor performer when the table gets more than half full, which is due to the phenomenon called *primary clustering*. Since we use the very first empty location

we come across after examining a number of occupied nodes, the groups of adjacent occupied addresses will inevitably tend to aggregate forming clusters. It means that if the table is quite full, we will eventually have to travel through the entire table trying to find an empty location to either insert a key or terminate searching. As a result, the entire performance advantage of even a relatively sparse table can be lost.

To alleviate the problem, linear probing can be replaced by some method of stepping through the table several nodes at a time in a pseudo-random manner. This manner, however, must be deterministic in the sense that for a given key hashing to a given address, the sequence of probes should be the same every time it is invoked. In addition, it should be able to examine all nodes in the table in an unbroken wrap-around cycle. This way, primary clustering is virtually eliminated: Since sequences are random, it is very unlikely that a new empty location will be chosen next to an occupied one.

To implement this plan, though, we have to select a *probe decrement* differently from letting it be just 1, as in linear probing. One way to do it is to hash the key, after it has already been hashed to find its hash address, one more time with a slightly different hash function using the result as a probe decrement for that particular key. Although the second computation takes some time, it usually leads to separate probing sequences for different keys spreading empty and occupied nodes evenly across the table.

The collision resolution policy just described is called *double hashing*. The question is, how to select the second hash function to satisfy the requirements formulated above? It follows from the number theory that if the probe decrement C and the hash table size HSIZE are *relatively prime*, C has the following property. If, starting at some address, we step through the table C nodes at a time in a wrap-around manner each node in the table will be covered exactly once. For instance, if HSIZE is prime, C can be any integer between 1 and HSIZE-1 inclusive, or if HSIZE is a power of 2 then C can be any odd integer between 1 and log(HSIZE)-1. In the program given below that implements double hashing for our sample file match problem, we still let HSIZE be prime leaving the rules of its selection intact, and compute C as

```
C = 1 + MOD(KEY, HSIZE-2) .
```

This has been proven to work satisfactorily in most cases. The code below presents a solution to our file match sample problem based on open addressing with double hashing, as opposed to coalesced linked list chaining:

```
*** OPEN ADDRESSING WITH DOUBLE HASHING ***;

DATA MATCH (KEEP=KEY S_SAT L_SAT);

ARRAY HKEY (0:&HSIZE) _TEMPORARY_;
ARRAY HSAT (0:&HSIZE) _TEMPORARY_;

DO UNTIL (EOF);
  SET SMALL END=EOF;
  H = MOD(KEY, &HSIZE);
  IF HKEY(H) = . THEN LINK INSERT;
  ELSE IF HKEY(H) NE KEY THEN DO;
    C = 1 + MOD(KEY, &HSIZE-2);
    IPROBE:
    H +- C;
    IF H < 0 THEN H ++ &HSIZE;
    IF HKEY(H) = . THEN LINK INSERT;
    ELSE IF HKEY(H) NE KEY THEN GOTO IPROBE;
  END;
END;
EOF = 0;
DO UNTIL (EOF);
  SET LARGE END=EOF;
  H = MOD(KEY, &HSIZE);
  IF HKEY(H) = . THEN FOUND = .;
```

```
ELSE IF HKEY(H) = KEY THEN FOUND = 1;
ELSE DO;
  C = 1 + MOD(KEY, &HSIZE-2);
  SPROBE:
  H +- C;
  IF H < 0 THEN H ++ &HSIZE;
  IF HKEY(H) = . THEN FOUND = .;
  ELSE IF HKEY(H) = KEY THEN FOUND = 1;
  ELSE GOTO SPROBE;
END;
IF FOUND THEN DO;
  S_SAT = HSAT(H);
  OUTPUT;
END;
END;
STOP;
INSERT:
HKEY(H) = KEY;
HSAT(H) = S_SAT;
RUN;
```

Once again, those revering "GOTOless" programming may try to make the code prettier without sacrificing any performance; it is not as easy as it may seem. Just like in the case of chaining, if the satellites are not needed the corresponding array and instructions referring to it should be omitted.

Let us take a look at the performance table given earlier. The results for double hashing were obtained using 50 percent sparse hash table with the load factor 0.5. Under these conditions, double hashing runs approximately at the level of separate chaining with 20 percent sparse table but consumes less memory. The memory advantage of double hashing is somewhat masked by the satellite array. Without the latter, it would be much more pronounced. So, we can conclude that this method of resolving collisions is quite fast; in fact, it even loads an equally sparse hash table faster than the separate chaining.

That a method based on stepping through the table before an empty node is encountered works so quickly, is to an extent amazing. However, this is a direct result of the double hashing methodology. In fact, experiments show that if the table is less than half full (or the load factor does not exceed 1/2), double hashing makes on the average no more than 2 comparisons in the case of unsuccessful search, and no more than 1.3 comparisons if the search is successful. Therefore, double hashing can be well recommended, especially if the memory is at stake.

NON-NATURAL KEYS

As we see, hashing performs admirably by any account, no matter what collision resolution policy is being employed. However, to this end, hashing has only expanded the applicability of key-indexing by eliminating the range restriction, but introduced another one. The assumption of keys being non-negative integers has been vital to both implementations. It seems like it is crucial if we want hashing to work properly, otherwise how would we hash the keys? Fortunately, the nature of the keys is not that critical because in its final stage, hashing is strictly comparison-based. Both hashes and traversals are used merely to minimize the number of comparisons necessary to carry out a search, yet the final decision, if a hash address is not empty, is made by comparing some keys in the table to the search key, anyway.

Therefore, in order to use hashing with keys of any nature, we only have to figure out how to hash a key if it is not a non-negative integer. Let us consider several different cases.

Fractional Signed Keys

In this case, we can simply rescale each key before hashing it by multiplying it by a suitable integer constant and adding another positive constant to the result if necessary. For instance, if our keys

can be represented as X.Y, multiplying each key by Y before hashing would suffice. If, in addition, they can be negative, we would simply add an integer Z known to exceed the largest absolute value a negative key can assume. So, the entire change to the programs above needed to accommodate such keys would be using

```
MOD ( KEY*Y + Z, HSIZE)
```

in the hashing formulae instead of the straight modulo. It will not cause any noticeable deterioration in performance, since in SAS this kind of computation is very fast.

Digital String Keys

In this case, it is a simple matter of using the INPUT function and an appropriate numeric informat. For example, if the keys were 16-digit account numbers stored as a character variable, we could simply choose

```
MOD ( INPUT (KEY,16.), HSIZE)
```

as our hash function. Accordingly, the hash table itself must now be organized as a *character* array:

```
ARRAY HKEY (0:&HSIZE) $16 _TEMPORARY_;
```

Character Keys in General

There are tons of papers written on the subject of hashing character keys. However, almost all of them are based on breaking a character key apart in some manner and then involving the individual bytes into a sort of computation resulting in an integer in the range [0:HSIZE-1]. Some of these methods, for instance, universal hashing, are actually quite good. However, they are based on the assumption that the process of extracting individual bytes from a string is very fast. Unfortunately, this is exactly what is slow in SAS. We would be much better off if we could convert a character string to an integer in a single shot. If the keys are no longer than 8 bytes, it is simple:

```
MOD ( INPUT (LEFT (KEY), PIB8.),
```

This method has an extra advantage: It avoids the slow SUBSTR function. Note that we use PIBw. instead of S370FPIBw. because it is faster, and in hashing the order of bytes does not matter: We only want to use as many key bytes as possible to minimize collisions. The LEFT function may help by squeezing leading blanks to the right. If the key is longer than 8 bytes, the trick will still work, provided that the 8 characters of the string following the blanks distinguish the keys well. However, if the first 8 bytes have a good chance of being identical, we will be forced to select the 8 bytes from the middle of the string or simply randomly, at the same time trying to keep SUBSTR usage low for the sake of speed.

Composite Keys

This situation arises quite often. A natural inclination to deal with such keys would be to concatenate them and then hash using one of the methods described above. Principally, there is nothing wrong about it. However, there are two pitfalls. First, concatenation is a slow operation - at least, in the context of hashing, where the speed of computing a hash function is of paramount importance. Second, even if the parts of a composite key are positive integers, it may happen that, when strung together, they will form an integer lying beyond the precision provided by the SAS numeric data type. Double-precision rounding typically begins after the 16th digit.

Consider a situation when our records are uniquely identified by two numeric variables, a 16-digit ID and 9-digit MEMNO, and a particular ID can correspond to many different accounts. If we converted both partial keys to character strings, concatenated them

as ID || MEMNO, converted the latter back to a numeric variable, and hashed it, the result would be quite miserable. Because of the rounding effect, we would in effect hash only the first 16 digits, that is, ID, while the entire MEMNO would be scrambled. That is, all composite keys with the same value of ID would hash to the same address regardless of the value of MEMNO! Aside from the overhead of the concatenation and conversions, that would lead to a large number of collisions killing performance.

Luckily, there is a good way around it. Remember, as far as hashing goes, we are not interested in the value of the key itself, but only in the remainder it would yield if divided by HSIZE. Horner's algorithm allows performing the following trick: Instead of creating a huge key (beyond SAS integer precision) and then hashing it, we can hash the key parts (represented accurately) separately, and then combine the results in the final hash address. The outcome will be exactly the same as would be achieved if we had enough integer precision to physically store the huge key accurately. For our hypothetical ID and MEMNO, it means that the hash function should be selected in the form:

```
MOD (MOD (ID,HSIZE)*1E9 + MEMNO, HSIZE) .
```

First, we get MOD(ID,HSIZE) never exceeding HSIZE-1, and multiply it by the range of MEMNO. For a hash table with 1E7 nodes, the result will not exceed 1E16. Then we add MEMNO. (It is peanuts compared to 1E16 but vital from the standpoint of hashing, since we are thus making the hash address depend on full MEMNO, too.) The result modulo HSIZE produces the hash address we need. If the partial keys are longer and/or their range is wider they can be, in turn, broken apart using division by a suitable power of 10 (but *not* conversion, unstringing, or concatenation of any kind!). After that, Horner's algorithm can be applied to the components.

Of course, in situations like this, the key parts should be stored in different parallel hash arrays, and loaded and tested separately. To be concrete, let us consider coalesced chain hashing. In our sample case with ID and MEMNO, we would replace the declaration of HKEY with

```
ARRAY HID (0:&HSIZE) _TEMPORARY_;  
ARRAY HMEM (0:&HSIZE) _TEMPORARY_;
```

and instead of HKEY(H) = KEY, populate them as

```
HID (H) = ID;  
HMEM (H) = MEMNO;
```

In the TRAVERSE subroutine, the equality test IF KEY = HKEY would have to be replaced with the instruction:

```
IF ID = HID(H) AND MEMNO = HMEM(H) .
```

A bit more extensive but similar modifications can be done in the implementation based on double hashing.

Summarizing, irrespective of the nature of the keys we have at hand, to hash efficiently, we should:

1. Involve as much of the information available in the keys in the hashing process to minimize collisions.
2. Use as few character operations as possible.
3. Make sure the result is always in [0:HSIZE-1] range.

APPLICATIONS

Subsetting is an important but only one of many tasks to which key-indexing and hashing can be applied successfully. However, before discussing other applications, we have to make a few final observations about subsetting, all the more that it is used as our proving grounds.

Subsetting

From the test results, it is evident that when it comes to subsetting, hashing (and especially key-indexing, if applicable) provides lookup speeds unmatched even by methods written in the underlying software and specifically designed for searching. As an icing on the cake, hashing is significantly more memory-efficient than formatting and SQL. The latter becomes extremely important when the number of keys in SMALL grows beyond a couple of million. The amount of memory used by hashing is directly proportional to the number of lookup keys and can be accurately estimated beforehand. But not with formats and SQL!

The author has been exposed to a number of production situations similar to the following true story. Once upon a time, in a powerful kingdom called MVS, the King imposed a 400 MB limit on a batch SAS job. Then His Majesty summoned two knights, Format and Hash, handed each two files, SMALL and LARGE, and ordered to match them. At 5 million records in SMALL, Format took about 20 CPU minutes to compile using almost all the memory, and then another 30 CPU minutes to match. Hash finished the entire job in 10 CPU minutes never using more than 160 MB. At 10 million records in SMALL, Format simply ran out of memory, yet Hash completed the task in 11 minutes using 320 MB of memory. (Of course, thereafter Hash was burned at the stake for sorcery, and Format married the Princess.)

On a different note, we should exercise certain caution when dragging satellites from SMALL through the memory. If there is more than one satellite, one may be tempted to create a separate parallel satellite array for each, but this is not always the right thing to do. Remember that temporary array items are allocated in 8-byte multiples per item. If we have four 8-byte character satellites, a separate array can be declared as \$8 for each satellite without wasting memory. However, if we have four 2-byte satellites and take the same path by declaring the arrays as \$2, it will not accomplish anything but waste of memory, for SAS will allocate the arrays with 8 bytes per item, anyway. So, in this case, we will be much better off memory-wise if we allocate one array as \$8 (or simply \$ defaulting to \$8), string the satellites together as the keys and satellites are loaded, and eventually unstring them on the way back just prior to output.

However, there might be an even better solution to handle satellites, especially if SMALL is, well, small. At the time of subsetting, we can forget about the satellites from SMALL and keep the keys only - hashing will run faster and consume at least 50 per cent less memory. After the subset is ready, it will be a simple matter to remerge or join it with SMALL obtaining all additional variables from SMALL we might need.

DATA Step Data Dictionaries

Let us take a look at key-indexing and hashing from a different standpoint. The key-indexed and hash tables we have used to facilitate direct address and hybrid searching can be viewed as *some abstract data type (ADT)* in memory, that allows to efficiently perform certain operations on its *entries*. The ADT used in key-indexing and hashing is simply called *a table*. The entries contain keys and maybe some satellite information. There are two operations we have learned how to perform in the process of solving our sample problem: *Insert and search*. Many kinds of ADTs other than a hash and key-indexed table can facilitate these operations. A simple sequentially searched array, binary searched sorted array, AVL tree are just a few ADT examples.

The difference between various ADTs lies in the time necessary to insert an entry or search the entries given a key. For example, a plain array requires $O(1)$, i.e. constant time, independent from the number of entries N , to insert a new entry - we simply append it to the right. However, searching such a structure occurs in $O(N)$ time, i.e. proportional to N . If the ADT is a sorted array, we need $O(N)$ time to insert an entry because it is necessary to shift a number of

items proportional to N to free a node for the new key keeping the table sorted. In exchange, searching an ordered array, as we know from Part 1, occurs only in $O(\log(N))$, or even $O(\log\log(N))$ time. Yet another ADT, an AVL tree, facilitates both operations in $O(\log(N))$ time as its worst case.

From these examples, it is clear what kind of advantage key-indexing and hashing offer: If a hash table is sparse enough, they support both insert and search operations in constant time $O(1)$, because, as we have seen before, it takes practically the same time to search the table or to insert a new key, no matter how many keys the table may contain.

As a side note, from this standpoint, the difference between key-indexing and hashing is merely superficial. A key-indexed table is, in effect, nothing else but an infinitely sparse hash table, and the hash function used to access it is simply constant.

The fact that hashing supports searching (and thus retrieval and update) in constant time makes it ideal for implementing a DATA step data dictionary. Imagine that in the course of DATA step processing, we need to memorize certain key elements and their attributes as we go, and at different points in the program, ask and answer questions like:

1. Has the current key element already been used before?
2. If not, how to insert it along with its attribute in such a way that the question 1 could be answered as fast as possible?
3. How to access a key element in the most speedy fashion and change its satellite data?
4. The key is no longer needed, how to delete it?

If the "key element" satisfies the conditions making key-indexing applicable, for instance, it is a SAS date, there is no better tool for the job. All the actions are performed in $O(1)$ time and do not get any simpler:

1. See if the node # KEY contains a missing value.
2. Fill the node with the attribute.
3. Overwrite the attribute already in the node.
4. Move a missing value to the node.

If the keys are not limited-range integers, we will have to organize a hash table using either of the collision resolution policies given in the text. In both programs, the body of the first DO UNTIL(EOF) loop constitutes nothing else but a ready-to-go combined hash search and insertion. That answers questions 1 and 2, or 1 and 3. The second DO UNTIL(EOF) loop is a pure hash search, and answers question 1 asked by itself.

A practical application of these principles immediately coming to mind is obtaining frequency counts in the case of a huge number of distinct levels of a categorical variable, when FREQ or SUMMARY either run out of memory or take too long to run. To compute frequencies without sorting, we must be able to maintain a table in memory allowing to immediately locate the value coming with the next record and add a unity to its count. The following question was asked in SAS-L by Alex Martchenko: "I have a credit policy SAS dataset with almost 100 million records. It has a numeric variable FLDR_ID that can be any integer number from -500,000 to +500,000. The dataset is unsorted by FLDR_ID. How to create a file with frequencies, cumulative frequencies, percents and cumulative percents for all values of FLDR_ID having only 50 MB of RAM?" The author gave the following key-indexing solution (a similar solution was offered by Ian Whitlock):

```
DATA DS_FREQ (KEEP=FLDR_ID FREQ CFREQ PCNT CPCNT);
  ARRAY F (-500001:500000) _TEMPORARY_;
  DO UNTIL(EOF);
    SET IDS END=END NOBS=NOBS;
    IF FLDR_ID = . THEN FLDR_ID = LBOUND(F);
    F(FLDR_ID) ++ 1;
  END;
  PTOT = 1/NOBS * 100;
```

```

DO I=LBOUND(F) TO HBOUND(F);
  IF F(I) = . THEN CONTINUE;
  FREQ = F(I);
  CFREQ ++ FREQ;
  PCNT = FREQ * PTOT;
  CPCNT ++ PCNT;
  IF I > LBOUND(F) THEN FLDR_ID = I;
  ELSE FLDR_ID = .;
  OUTPUT;
END;
RUN;

```

This program takes about 12 MB of memory and runs at least an order of magnitude faster than either `FREQ` or `SUMMARY` (if they *do* have enough memory).

Sortless unduplication

Discussing hashing, we saw that as an attempt is made to load the next key into a hash table, the search-and-insert routine first determines whether the key has already been inserted, and if it has, goes to the next record. This side effect of hashing can be used to unduplicate files.

Normally, duplicates are removed as a side effect of `PROC SORT`. However, if sorting is not needed, a lot of extra work is done for no reason. Another method, SQL with the `DISTINCT` keyword, is "pure", but in this case the value supplied to `BUFFERSIZE` has no effect, and the procedure runs slowly.

To capitalize on the unduplicating side effect of hashing all we will have to do is make it pure: Load the hash table and then simply output its contents skipping empty nodes. For instance, if we select double hashing, we will only have to delete `L_SAT` from the `KEEP` statement, and replace the second `DO UNTIL` loop with the following code excerpt:

```

DO H=0 TO &HSIZE;
  IF HKEY(H) = . THEN CONTINUE;
  KEY = HKEY(H);
  S_SAT = HSAT(H);
  OUTPUT;
END;

```

If we run the code with this change (do not forget that the `%LET` statement and `DATA` step defining `HSIZE` must still be in place!), the file `MATCH` will now contain an unduplicated version of `SMALL`. For example, for `SMALL` consisting of 1,000,000 records with 16-digit numeric keys, 10,000 of which are unique, the duplicates get removed in about 8.3 seconds. An excerpt from the SAS log below shows how other methods fare in comparison:

```

NOTE: The DATA statement used 8.27 seconds.
NOTE: The PROCEDURE SORT used 16.95 seconds.
NOTE: The PROCEDURE SQL used 29.12 seconds.

```

It should make the comparison quite clear.

CONCLUSION

Key-indexing is an array lookup technique based strictly on direct addressing into an array. Its area of applicability is limited to integer keys falling in a limited range defined by available memory resources. However, when applicable, key-indexed search exhibits unmatched performance, and is the most straightforward way of implementing an ADT where all operations, such as search, insert, retrieve, update, delete, and enumerate are done in constant time.

Hashing expands the idea of key-indexing making the direct addressing approach possible for any keys. It is achieved by resorting to a hybrid strategy combining direct addressing with sequential searching. Although somewhat slower than key-indexing,

hashing nevertheless offers tremendous performance advantages compared to any other search method including SAS formats and SQL, while consuming significantly less memory.

Key-indexing and hashing are cool. They allow a SAS programmer to easily operate in the niches where other approaches either run out of memory or take too long to run. The author encourages other SAS users to use these tools, modify them, tweak them, improve the code, and discover new areas of application.

As Karsten Self wrote once, "Hash rocks, dude!" I hope that after you try these methods in your practical SAS activity and find what they are capable of, you will agree.

SAS is a registered trademark or trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

REFERENCES

1. D. E. Knuth, *The Art of Computer Programming*, **2**.
2. D. E. Knuth, *The Art of Computer Programming*, **3**.
3. R. Sedgewick, *Algorithms in C*, Parts 1-4.
4. T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.

ACKNOWLEDGEMENTS

Special thanks to Karsten M. Self and F. Joseph Kelley for their enthusiastic support of hashing applications in SAS and valuable discussions full of ideas, wit, and vigor. Thanks to Don Stanley who prompted the author to look beyond pure key-indexed search. The author would also like to acknowledge the contributions of the following individuals:

Doris H. Bogar	Michael V. Dorfman
Victor P. Dorfman	Colin Earle
Yuri Katsnelson	Vladimir A. Kirillov
Eugenia P. Kravchenko	Alex V. Martchenko
Michael A. Raithel	Vera M. Voloshin
Ian Whitlock	Matthew A. Wilson

AUTOR CONTACT INFORMATION

Paul M. Dorfman
 10023 Belle Rive Blvd. 817
 Jacksonville, FL 32256
 (904) 564-1931
 sashole@earthlink.net

